APPLICATION FOR
UNITED STATES PATENT
IN THE NAMES OF

Stephen Anthony Edwards

For

Method and Apparatus For Converting A
Concurrent Control Flow Graph Into A
Sequential Control Flow Graph

DOCKET NO. 4000.005 (A1999-039)

Please direct communications to:
Brown Raysman Millstein Felder & Steiner
120 West 45th Street
New York, NY 10036
(212) 944 - 1515
Express Mail Number EJ593691494US

**Method and Apparatus For Conv rting A
Concurr nt Control Flow Graph Into A
S qu ntial Control Flow Graph**

FIELD OF THE INVENTION

5
The present invention relates generally to the conversion of concurrent program specifications into an equivalent sequential program specification, and more specifically to the conversion of control flow graphs.

BACKGROUND OF THE INVENTION

In the utilization of concurrent programming languages, it is often
10
desirable to be able to convert a concurrent specification of a program into an equivalent sequential specification (for efficiency reasons, in order to be run on a sequential processor).

For example, real-time embedded computer systems (or more generally, reactive real-time systems) are often most effectively specified, from a functional
15
level, in terms of a concurrent programming language. In terms of providing an executable for the embedded system, which can be executed with greater efficiency (in terms of both speed of execution and/or hardware resources), it is often most effective to provide a sequential executable that provides equivalent functionality to the current functional specification.

20
A very general form of concurrent programming is the specification of a concurrent control flow graph. A very general form of sequential programming is the specification of a sequential control flow graph. It is therefore desirable to have efficient procedures for converting a concurrent control flow graph into a sequential control flow graph.

25
A control flow graph (or CFG) is essentially a kind of flow chart, as that term is conventionally understood, that depicts the flow of control of a program as edges connecting nodes (where the nodes represent operations to be performed). Since these edges define possible flows of control, they may be referred to as "control edges." The nodes of a CFG comprise plain and

conditional nodes, each with an expression. When control reaches a node, the node's expression is evaluated and control flows along one or more edges leaving the node. A plain node has a single outgoing edge and its expression is usually an assignment. Control leaves a conditional node along the edge whose label matches the value of the expression.

In a sequential CFG (or SCFG) there is only a single path of execution (or thread) through the CFG.

A concurrent CFG (or CCFG) is a CFG which also includes fork and join nodes, each of these nodes also having an expression. Fork and join nodes start and collect groups of parallel threads. Control flows out all edges leaving a fork, starting a group of threads that will wait at a matching join node before continuing. Fork and join nodes may nest, but control may not pass between threads. Specifically, all paths from a particular fork must meet for a first time at a unique join.

In addition to specifying a concurrent program in terms of a CCFG, it is often desirable to express the concurrent program in a higher-level programming language which is then translated into a CCFG. Alternatively, it may be desirable to express the concurrent program in terms of a graphical language that is then translated into a CCFG.

An example of a suitable concurrent programming language for specifying the functionality of an embedded computer system is the Esterel language. The Esterel language is described in Berry and Gonthier's "The Esterel Synchronous Programming Language: Design, Semantics and Implementation," Science of Computer Programming, volume 19 number 2 pages 87-152, November 1992 (Elsevier Science, Amsterdam, The Netherlands), which is herein incorporated by reference. This paper formally describes the semantics of the language.

Esterel has the control constructs of an imperative language like C, but includes concurrency, preemption, and a synchronous model of time like that used in synchronous digital circuits. In each clock cycle, an Esterel program restarts, reads its inputs, and determines its reaction.

Figure 1 shows a simple Esterel program with two concurrent threads. The first thread waits for the START signal and emits REQUEST. If it receives GRANT in the same cycle, it emits the GOT signal. In alternating cycles, the other thread emits GRANT in response to REQUEST. The threads restart when

5 the RESET signal appears because of the abort-when RESET construct inside the outer loop.

The translation of Esterel into the CCFG may be accomplished in a variety of ways. A particular translation of Esterel into a CCFG is presented herein by pairing Esterel statements with their corresponding implementation as a program

10 fragment of "concurrent C." Concurrent C is a form of pseudo-code which is utilized in this patent for expository purposes. Concurrent C is essentially the same as standard C, with the addition statements that perform the fork and join functions. The translation of concurrent C into a CCFG can be accomplished by a variety of known methods. In this patent, Esterel statements are paired with

15 their corresponding concurrent C program fragment, rather than with their corresponding CCFG fragment, for expository convenience.

Figures 2A through 2D depict the translations of the Esterel statements, that do not affect time, into concurrent C code. For each of these Figures, the Esterel statement is on the left and the concurrent C code translation is on the

20 right.

The "exit" statement of Esterel throws an exception that can be caught by a surrounding "trap T in ... handle T do." This only happens after all threads in the same group are done for the cycle. To handle this, each thread sets an exit level when it stops at the end of a cycle. This level indicates termination

25 (level 0), pausing (level 1), or an exception (levels 2 and higher). Exceptions take precedence over pauses, so a group of threads responds only to the highest level.

A "pause" statement resumes in the next cycle. A pause statement is shown on the left side of Figure 3A, with the right side depicting its translation

30 into concurrent C. The operation of this concurrent C is as follows. The code sets its threads state to "k," making the "switch" statement surrounding the

thread send control to the "case" label next cycle. Raising the exit level to 1 indicates this thread has paused. The branch to "join" stops the thread for the cycle.

The "await" statement is similar to "pause," but it also pauses in later
5   cycles until its signal is present. An await statement is shown on the left side of Figure 3B, with the right side depicting its translation into concurrent C.

Esterel's preemption statements, such as "abort," introduce the equivalent of nested "switch" statements in concurrent C. This is shown in Figure 3C where an Esterel abort statement is shown on the left and is paired with its translation
10   into concurrent C on the right side. In the first cycle, "abort" just runs its body. It restarts its body in later cycles only if the aborting signal is absent.

The Esterel "suspend" statement runs its body in the first cycle and pauses in later cycles when the suspending signal is absent, leaving its thread's state unchanged. This operation of suspend is depicted in Figure 3D where an
15   Esterel suspend statement is shown on the left and is paired with its translation into concurrent C on the right side.

The Esterel "signal" statement creates a new, absent copy of its signal. This operation of signal is depicted in Figure 3E where an Esterel signal statement is shown on the left and is paired with its translation into concurrent C
20   on the right side.

The Esterel "exit" statement raises its process's exit level to two or more depending on the exception. Since this terminates the thread and its process, there is no need to set the thread's state. This operation of exit is depicted in Figure 3F where an Esterel exit statement is shown on the left and is paired with
25   its translation into concurrent C on the right side.

In Esterel, "parallel" and "trap" statements are intertwined. An example of this is shown on the left side of Figure 3G. An implicit trap surrounds each group of parallel threads, and the body of a trap is considered a separate thread. The trap/parallel combination resets the exit level for the enclosed process, runs the
30   threads within, and handles the exit level they return. In the concurrent C translation of Figure 3G, the process terminates if the level is zero (the switch

falls through), pauses at level one, and handles exceptions at levels two and higher.

Furthermore, the concurrent C translation of the right side of Figure 3G implements the Esterel "parallel" and "trap" statements as follows. The threads
5   have two entry points: one taken in the first cycle, the other taken in later cycles that use "switch" statements to restart the threads. The "fork" statement passes control to each of its labels. The "join" waits until all the threads branch to it before continuing. A terminated thread sets its state to zero so control will go to the case 0: labels when other threads in the process continue to run.

10   In addition to the above pairings, the translation of Esterel nested abort statements into an CCFG is accomplished according to the following procedure. The following procedure also applies to the simpler situation of sequenced pause statements.

At the beginning of each clock cycle, every running Esterel thread checks
15   the signals that might abort running blocks before resuming where it paused in the last cycle. Each CCFG thread simulates this behavior by saving its state at the end of a cycle and resuming at the beginning of the next with "switch" statements.

Nested aborts in the Esterel program are handled with nested switches in
20   the concurrent C code. A thread's aborts form a tree with a signal at each node and a pause or group of threads at each leaf. Restarting a thread at the beginning of a cycle requires checking for abortion signals along the path from the root to the leaf that had control at the end of the last cycle.

Each node of the tree is translated into an "if" that checks the aborting
25   signal and a "switch" that sends control to a child. The encoding of a thread's states (which corresponds to the leaves of the preemption tree) simplifies the decision at each switch. The edges leaving each node are numbered 0, 1, 2, etc. and become the "case" labels. The sequence of edge labels from the root to a leaf becomes the encoding for the leaf. These labels are packed into a single
30   machine word so each switch statement can extract them with a mask and a shift. Figure 4 depicts an example that illustrates state encoding with nested

aborts (and sequenced pauses) on the left in Esterel and the translation into state-encoded concurrent C on the right. Code that sets exit levels (explained above) is not shown in Figure 4.

5   **SUMMARY OF THE INVENTION**

The present invention accepts an acyclic concurrent control-flow graph and produces a sequential control flow graph that, when executed, behaves functionally like the CCFG would if it were run on concurrent hardware. An SCFG can be easily translated into a traditional sequential programming language such 10   as C or assembly to be executed on a traditional sequential processor.

Determining the order in which CCFG nodes will be run is the first step in the process. Control edges in the CCFG constrain the order in which CCFG nodes must run; communication between threads generally impose further constraints. For example, the Esterel language requires all statements that write 15   a variable run before any statement that reads the same variable.

An easy way to further constrain a valid order of CCFG nodes is to augment the CCFG with data dependence edges (representing inter-thread communication) and topologically sort the nodes in the augmented graph.

Once the CCFG nodes are ordered, the procedure for producing the 20   SCFG from the scheduled acyclic CCFG simulates the execution of the CCFG under an operating system supporting concurrent threads and creates an SCFG that, when executed, will reproduce the functional behavior of the CCFG running under this simulated operating system. The effects of context switching are largely compiled away by this simulation process. Each context switch is done 25   by a single assignment that stores the state of the thread being suspended and a single branch that restores the state of the thread being resumed.

In particular, the procedure produces the SCFG by stepping through the CCFG nodes in scheduled order, copying each node and its incoming edges to the SCFG. During this simulation, the procedure maintains a set of currently 30   active (i.e., runnable) threads. When a node is encountered during simulation that does not reside in one of the active threads, or when two nodes in different

active threads appear one after the other, the procedure adds nodes to the SCFG that perform a context switch. These save the state of a running thread and resume the thread of the current node with a multi-way branch.

The procedure handles fork and join nodes specially. These nodes
5   represent starting and terminating groups of threads. When a fork node is encountered, a new thread is created for each of its outgoing edges. These threads begin in a suspended state, forcing a context switch whenever the first node in any of the threads is encountered. When a join node is encountered, each of its threads is terminated.

10   While the present invention minimizes the cost of context switches, they are still relatively expensive. Minimizing context switches through a careful choice of order is desirable, but appears to be NP-complete. Fortunately, experiments suggest a simpleminded topological sort will produce acceptably efficient schedules, although heuristic search techniques could be applied to
15   improve schedule quality.

The present invention requires the edges in the CCFG to be acyclic to ensure that there exists an order of the nodes such that all edges are forward. This does restrict the class of systems the present invention is able to generate SCFGs for, but many useful systems have this acyclic property.

20   Advantages of the invention will be set forth, in part, in the description that follows and, in part, will be understood by those skilled in the art from the description or may be learned by practice of the invention. The advantages of the invention will be realized and attained by means of the elements and combinations particularly pointed out in the appended claims and equivalents.

25

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, that are incorporated in and constitute a part of this specification, illustrate several embodiments of the invention and, together with the description, serve to explain the principles of the invention:

30   Figure 1 shows a simple Esterel program with two concurrent threads;

Figures 2A through 2D depict translations of Esterel statements, which do not affect time, into concurrent C code;

Figures 3A through 3F depict on the left, respectively, Esterel statements pause, await, abort, suspend, signal and exit; on the right Figures 3A through 3F

5   depict the translation of the Figure's Esterel statement into concurrent C code;

Figure 3G shows, on the left, the intertwined usage of Esterel trap and parallel statements, while the right side of this Figure shows the translation of such trap and parallel statements into concurrent C;

Figure 4 depicts an example that illustrates state encoding with nested

10   aborts (and sequenced pauses) on the left in Esterel and the translation into state-encoded concurrent C on the right;

Figure 5 illustrates on the left an Esterel program that can execute a statement multiple times whereas the right side of Figure 5 shows concurrent C (with the parallel operator of Esterel) which eliminates multiple execution of

15   statements by making multiple copies of them;

Figures 6A through 6I show detailed pseudo-code for the translation process from a scheduled acyclic CCFG to an SCFG;

Figure 7 presents an exemplary scheduled acyclic CCFG for input to the pseudo-code of Figure 6;

20   insert 161 ⟩ Figures 8A through 8N and 8P through 8Q depict the simulated execution of the pseudo-code of Figure 6, upon the example CCFG of Figure 7;

Figure 9 depicts a hardware environment in which the present invention can be operated; and

Figure 10 depicts an overview flowchart of the CCFG to SCFG conversion

25   process.

BRIEF DESCRIPTION OF THE APPENDIX

The accompanying Appendix, that is incorporated in and constitutes a part of this specification, illustrate several embodiments of the invention and, together

30   with the description, serve to explain the principles of the invention:

Appendix A presents the detailed step-by-step simulated execution of the pseudo-code of Figure 6 upon the example of Figure 7.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

5   Reference will now be made in detail to preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts.

The present invention accepts an acyclic CCFG as input and produces an
10   SCFG as output. Once an SCFG has been produced, a variety of known methods can be used to translate the SCFG into a non-concurrent language (such a C or assembler) which can be sequentially executed by a computer system. The translation of an acyclic CCFG into an SCFG is accomplished as follows.

15   An CCFG differs from an SCFG in that it has concurrently executing threads. If these currently executing threads have no data interaction among them, then the translation from CCFG to SCFG is straightforward: the currently executing threads can be placed "end to end" in a sequential order.

If, however, there is at least one bidirectional communication, between at
20   least two threads, then the translation from CCFG to SCFG is considerably more complex. For example, if there is a thread1 and a thread2, which are concurrently executing, and in which the following occurs: i) data produced by thread1 (a "writer"), ii) the data produced by thread1 is considered by thread2 (a "reader"), iii) data is then produced by thread2 (now thread2 is a "writer") and iv)
25   the data produced by thread2 is considered by thread1 (thread1 is now a "reader").

Scheduling the nodes in the CCFG is the first step in producing a SCFG from it. A schedule is a total order on the nodes. Technically, for each pair of nodes, one is considered "earlier," the other "later." A typical way to describe this
30   total order is to give each node a small integer label. For example, the first node is labeled 1, the second 2, and so forth.

The present invention requires the nodes in the CCFG to be ordered such that every control edge is forward, that is, if there is an edge between two nodes, the source node is earlier than then destination node. The communication semantics of the source language usually imposes additional constraints on the

5   order. For example, to implement Esterel semantics, a data edge can be added leading from each node that writes a variable to each node that reads the same variable.

One way to further constrain a valid order for the nodes is to augment the CCFG (which has thus far been presented as only containing control edges) with

10   data dependency edges (or simply "data edges") representing communication constraints and topologically sort the augmented graph. As with a control edge, no data edge passes from a node with a later number in the order to a node with an earlier number in the order. For example, to implement Esterel semantics, a data edge can be added leading from each node that writes a variable to each

15   node that reads the same variable. Figure 7 depicts a scheduled ACCFG where each node has already been assigned an integer (from 1 to 8) to indicate its location within the order. All the of the edges in Figure 7, depicted with straight lines, are control edges. There are two edges in Figure 7, specifically the edge from node 3 representing the statement "emit B" to node 4 representing the

20   statement "conditional B" and the edge from node 5 representing the statement "emit C" to node 6 representing the statement "conditional C," that depict data edges. As can be seen, these two data edges are distinguished by being drawn with jagged lines.

A topological sort of an augmented CCFG is one way to determine a valid

25   ordering (or scheduling) of the nodes in a CCFG. This well-known procedure recursively visits each node in the graph and adds the node to the beginning of the topological order after visiting all of the node's successor nodes. The resulting order can easily be shown to have the property mentioned above, that is, every edge in the CCFG will lead from an earlier node to a later one.

30   Whenever a node from one thread is followed immediately by a node from another thread, the present invention produces a relatively expensive context

switch in the SCFG. Minimizing these context switches is desirable, but appears to be an NP-complete problem (it is as hard as the minimum feedback vertex set problem). Experiments suggest that using a topological sort to order nodes produces orders with acceptable numbers of context switches. However,

5 heuristic search techniques could be applied to further reduce their number.

The present invention requires the edges in the CCFG to be acyclic to ensure that there exists an order of the nodes such that all edges are forward. This does restrict the class of systems the present invention is able to generate SCFGs for, but many useful systems have this acyclic property.

10 The translation of Esterel programs described in the background section occasionally produces cyclic CCFGs for Esterel programs that should have a sequential implementation. The cycles in these CCFGs can be removed using a procedure based on the one described by Gerard Berry in "The Constructive Semantics of Esterel," a book in preparation available at

15 http://www.inria.fr/meije/esterel/, and herein incorporated by reference. Berry's procedure was originally designed to cure a similar problem that arises when Esterel is translated into a circuit netlist as in his V4 and V5 compilers. The V4 and V5 compilers are described in the Gerard Berry book, as well as in "Esterel on hardware," by Gerard Berry, Philosophical Transactions of the Royal Society

20 of London, Series A, 339:87-104, 1992, that is herein incorporated by reference.

The example in Figure 5 shows an Esterel fragment that produces a cyclic CCFG when translated using the procedure described earlier. The problem comes from the "present" statement, which can run twice in a cycle. When "exit T" runs, it terminates the trap, causing the outer loop to immediately restart the

25 threads. This causes "present A" to run again. The right side of Figure 5 shows how this problem can be dealt with by making a separate copy of each invocation of the present statement.

In fact, it is possible to write an Esterel program that executes certain statements many times within a single cycle. All are roughly of the form shown

30 in Figure 5. The problem arises when the first block of code within a trap within a loop can be interrupted and restarted within the same cycle. The cycle involves

the code that handles exceptions, which appears to be able to send control back around the loop indefinitely. However, the Esterel language requires that all loops must pause for at least one cycle, so this cyclic control path can only be taken a finite number of times within a cycle. The solution is to duplicate the first

5 block of code, which is guaranteed not to instantaneously terminate and restart itself. The result of such duplication is shown in the right side of Figure 5.

The cycle of Figure 5 is a side effect of Esterel semantics and may not appear in other languages. The procedure for unrolling these cycles is specific to Esterel and may not be applicable to other programming languages.

10 Certain Esterel programs cannot be converted into an acyclic CCFG, and can only be expressed as a cyclic CCFG. These include programs where the required order of execution of two or more statements may change depending upon the particular data to be processed. For example, there can be a statement1 in a thread1 of an Esterel program and a statement2 in a

15 concurrently running thread2 of the same Esterel program. Such a program may have the required order of execution of statement1 and statement2 change depending upon the data to be processed.

Although the present invention has many uses, one possible use is as one of a series of steps that together translate Esterel into sequential C code. For

20 example, a translation might begin by translating Esterel into a possibly cyclic CCFG using the procedure described in the background section. Next, any cycles in the CCFG are removed by the procedure discussed above that duplicates nodes in the CCFG. Once the CCFG is acyclic, it is passed to the present invention, along with an ordering for the nodes, which then generates an

25 SCFG. As mentioned above, if scheduling is included as a step of the present invention, any one of a variety of scheduling approaches could be used, including the topological sort procedure described above or a heuristic scheduler. Finally, the following simple procedure can be used to translate the SCFG into a conventional sequential code (such as C or assembler): order the

30 nodes in the SCFG, generate code for each, and place goto statements where necessary to produce the needed flow of control. Any of the steps surrounding

the invention could be modified significantly to accommodate a different language, different mechanisms for removing cycles, and so forth, without change to the invention.

5   The present invention is also described in the following paper: Stephen Edwards, "Compiling Esterel into Sequential Code," Proceedings of the 7th International Workshop on Hardware/Software Codesign, Rome, Italy, May 3-5, 1999, pages 147-151, published by the Association for Computing Machinery, herein incorporated by reference. This paper includes a discussion of a variant of the CCFG presented here.

10   The translation process, from a scheduled acyclic CCFG to an SCFG, is described, in detail, in the pseudo-code of Figure 6. The pseudo-code of Figure 6 is loosely based on the C and C++ programming languages.

An exemplary scheduled acyclic CCFG is depicted in Figure 7.

The simulated execution of the pseudo-code of Figure 6, upon the

15   example CCFG of Figure 7, is depicted in Figure 8 and is also described in Appendix A.

Before discussing the detailed procedure of Figure 6, it is useful to present a general overview of it. In general, the procedure for producing the SCFG from the scheduled acyclic CCFG is as follows. Each node of the CCFG

20   is looped over, in scheduled order, such that it becomes the "current node" (or "cn"). A copy of the current node is produced for the SCFG, called the "current SCFG node" (or "sn"). If the thread in which the current node resides is suspended and not currently running, a context switch is inserted between the nodes in the SCFG representing the previously-running thread and the current

25   SCFG node. Regardless of whether the thread of the current node is running or not, edges to the current SCFG node are added, such edges leading from the current node's predecessors in the SCFG and reflecting the current SCFG node's placement within its thread which must now be running. This process is also depicted as a flowchart in Figure 10.

30   A context switch consists of two parts: nodes that save the state of the thread that was executing previously (that is, the one being suspended), and a

conditional node (a "restart node") that tests the saved state of the node being resumed (that is, the thread in which the current node resides). This state was saved by the assignment nodes of a previous context switch when the thread was suspended. The conditional node branches to the current SCFG node if the

5   saved state of the thread matches the ID of the node (usually the number of the current node in the schedule), indicating that the current node was about to run when its thread was last suspended. Once the context switch has been inserted, subsequent current SCFG nodes may also represent additional continuation points of the thread just resumed (provided there is no intervening context

10   switch) and each of these continuation points will also be branched to, based on the previously-saved state, from the same restart node.

As discussed above, context-switching code in the SCFG saves the state of suspended threads in state variables. It should be noted that the procedure of the present invention for constructing the SCFG manipulates the names of these

15   variables, but not their values. The values of these variables are only needed when the SCFG is running, after it has been generated.

Subsequent to the process of Figure 10, the resulting SCFG is typically translated, by known techniques, into a programming language description reflecting the SCFG's sequential thread of control. This programming language

20   description is then, typically, transferred to the target architecture of an embedded system which executes it. The SCFG resulting from the process of Figure 10 may be executed in other ways as well. For example, the SCFG could be interpreted directly.

25   ## 1. PSEUDO-CODE DESCRIPTION

The pseudo-code of Figure 6 is structured as follows.

The main data types of an ACCFG are: cnode, process and thread. The main data type of the SCFG is the snode. Each snode has an expression that is usually copied from its counterpart in the CCFG and a set of outgoing edges,

30   each of which may have a label. Figure 6A defines the cnode data type, while Figure 6B defines the process and thread data types.

A cnode represents all nodes of the ACCFG. For example, each node of the exemplary ACCFG of Figure 7 would be represented by a cnode data structure. In the process of converting the ACCFG into an SCFG, the input ACCFG is augmented with the additional data structures of the process and

5   thread. A process is defined as a class extension of the cnode class and therefore inherits its properties. In the course of converting the ACCFG into an SCFG, a process is created each time a fork node is encountered in the ACCFG. The basic purpose of the process is to contain, and keep track of, the plurality of concurrent threads which the fork node starts. Each of these concurrent threads

10   is represented by the thread data structure and these thread data structures are placed within the process data structure representing the fork node which started the thread.

Figures 6C through 6E depict the main routine for converting an ACCFG into an SCFG. This routine is entitled "synthesize a scfg" and it produces an

15   SCFG as output when given a scheduled ACCFG as input. In particular, "synthesize a scfg" produces an SCFG which has a single entry node (the first node to be executed in the SCFG) and a single exit node (the last node to be executed in the SCFG). The procedure expects the ACCFG to have a single exit node, which will become the SCFG exit node.

20   Figure 6C depicts the block of initialization assignment statements which create the initial process and thread data structures by which conversion of the ACCFG into an SCFG can begin. Specifically, an outermost process (called "op") is created. Figure 6C, line 12. This is the only process which does not correspond to any fork node. Within the outermost process the outermost thread

25   (called "tt") is created. The single entry node for the SCFG (called "en") is created. Figure 6C, line 10. Also, the first scheduled node of the input ACCFG (obtained at Figure 6C, line 32) is put in the outermost thread. Figure 6C, lines 37 and 39.

The main loop of "synthesize a scfg" begins at Figure 6D. The main "for"

30   loop iterates over each cnode of the input ACCFG, in scheduled order. Figure 6D, line 4. Specifically, each cnode of the input ACCFG becomes the current

cnode and is assigned to the variable "cn." The first action of the main loop is to make an snode copy of the current cnode for the SCFG graph. Figure 6D, line 6. This snode copy of the current cnode is called the current snode and is assigned to the variable "sn." Next, the thread in which the current cnode is

5 contained is determined. Figure 6D, line 7. Note that for the very first scheduled cnode of the ACCFG, the initialization code of Figure 6C has already placed it within the outermost thread.

The rest of the body of the main loop is divided into four main code blocks labeled A, B, C and D. Code blocks A and B are shown in Figure 6D, while code

10 blocks C and D are shown in Figure 6E. For each current cnode, a code block from A or B, and a code block from C or D is executed. Specifically, for a fork node code blocks B and C are executed. For a join node code blocks A and D are executed. For all other current cnodes (also referred to as "normal" cnodes), code blocks B and D are executed. This execution of code blocks can be viewed

15 as follows. Usually, for normal cnodes, code blocks B and D are performed. When a fork is encountered, the same code blocks are executed as for a normal cnode, except that code block C is substituted for code block D. Similarly, when a join is encountered, the same code blocks are executed as for a normal cnode, except that code block A is substituted for code block B. This execution of code

20 blocks depending upon cnode type is depicted in tabular form in the comments of Figure 6D at lines 16-18.

Figures 6F through 6I depict the support functions for "synthesize a scfg." Specifically, Figure 6F depicts the support functions "run cnode cn as snode sn" and "put cnode cns in thread th." The input parameters to "run cnode cn as

25 snode sn" are "cn" to which a cnode is passed and "sn" to which an snode is passed. The input parameters to "put cnode cns in thread th" are "cns" to which a cnode is passed and "th" to which a thread is passed. Figure 6G depicts the support function "switch to thread th." The input parameter to "switch to thread th" is "th" to which a thread is passed. Figures 6H and 6I show the support

30 function "suspend any running thread in process p." The input parameter to "suspend any running thread in process p" is "p" to which a process is passed.

2. SIMULATED EXECUTION OF PSEUDO-CODE

The simulated execution of the pseudo-code of Figure 6, upon the input ACCFG example of Figure 7, is discussed below. It should be noted that this

5   simulated execution of the pseudo-code of Figure 6 is not the simulation objective of the pseudo-code itself. Once the pseudo-code of Figure 6 is translated into actual code, execution of the actual code results in simulating (as discussed above in the Summary of the Invention) the execution of the input CCFG under an operating system supporting concurrent threads.

10   Appendix A presents a step-by-step pseudo-code listing of the simulated execution of the pseudo-code of Figure 6 upon the example of Figure 7. In conjunction with Appendix A, Figure 8 presents, graphically, the simulated execution of the example of Figure 7 on the pseudo-code of Figure 6. While Appendix A and Figure 8 are detailed, they are not exhaustive presentations of

15   every simulated step of execution of the pseudo-code of Figure 6.

The underlined headings of Appendix A denote the execution of certain key code blocks of Figure 6, or the entry of certain key functions of Figure 6.

Appendix A begins with the heading "synthesize a scfg" indicating that the pseudo-code function "synthesize a scfg" has been called (Figure 6C, line 4) with

20   the ACCFG of Figure 7 as its input parameter.

2.1 INITIALIZATION

The next heading of Appendix A is "Initialization." This indicates the execution of the block of pseudo-code statements, within "synthesize a scfg,"

25   that are titled with the comment "INITIALIZATION." Figure 6C, lines 6-41. Immediately below the Appendix A heading "Initialization," is a comment stating "See Figure 8A." This comment indicates that from its point forward in Appendix A, until the reference to "See Figure 8B" is reached, the step-by-step execution of Appendix A should be viewed in conjunction with the graphical illustration of

30   Figure 8A.

In general, the function of the "INITIALIZATION" block of pseudo-code statements is to set up the outermost process and put a single outermost thread within it. This outermost thread starts out suspended. The execution of the first iteration of the "MAIN LOOP" (discussed below) will resume this outermost

5   thread such that the topologically first node of the input ACCFG can be converted into the output SCFG. The step-by-step execution of the "INITIALIZATION" block, upon the particular example of Figure 7, is presented below.

        As can be seen in Appendix A, the first action under the "Initialization"

10   heading is the creation of an entry node (called "en") for the SCFG being created with the node being assigned the number 1003.

        Next, the outermost process (called "op" and indicated by the number 1000) is created.

        Subsequent to creating process 1000, as can be seen in Appendix A, this

15   process is referred to in the listing of Appendix A as "process_1000." The statement "process_1000.state = Runnable" indicates that the "state" property of process_1000 is being assigned the value "Runnable." Setting the state of process_1000 to "Runnable" indicates that process_1000 can run one of the threads of control it contains, but that currently none of its contained threads is

20   running.

        The "runningThread" property of process_1000 is set to "none" since there is no thread inside it currently running.

        Next, entry node 1003 is added to the "runningPredecessors" of process_1000, and the pointer from process_1000 to entry node 1003 is given

25   the "empty" label. This is accomplished by the pseudo-code of Figure 6C, lines 18-20. Note that the "+=" operator means, in general, that the item on the right-hand-side of the operator is being "added" to the collection of like items on the left-hand-side of the operator. In this particular case, the "+=" operator means that process_1000 could point to multiple nodes as being among its

30   runningPredecessors. The expression on the right-hand-side of the "+=" operator of Figure 6C, line 18, namely "(en, -)", indicates both what node

process_1000 points to, by way of pointer 1065, and the label to be associated with the pointer 1065. In general, for an expression "(<leftarg>, <rightarg>)," the argument <leftarg> specifies the node to be pointed to while <rightarg> specifies the label for the pointer. If <rightarg> is a hyphen (i.e., "-"), then the

5   empty label is specified.

In general, the values of labels are only compared when attached to the outgoing pointer from a conditional node. For conditional nodes, the label indicates that the pointer (also known as "edge") it is associated with should be taken when the expression of the conditional node evaluates to the value

10   specified by the label. If the label of a pointer is empty, then that pointer becomes the default pointer of the conditional node it originates from. The default pointer from a conditional node is taken if none of the other pointers from the conditional node, all with non-empty labels, match the value of the conditional node's expression.

15   In Figure 8A, pointer (or edge) 1065 has its empty label indicated by the expression "label = empty." In Figure 8, an alternative way of denoting an edge as having an empty label is to provide the edge with no label indication. Edge 1065 has been provided with the abbreviation "runP" to indicate that it points to a "runningPredecessor" of process 1000. In general, the use of the abbreviation

20   "runP" in Figure 8 indicates that the edge points to a runningPredecessor. Similarly, other edges in Figure 8 have the abbreviation "resP" which indicates that the edge points to a "restartPredecessor."

At this point it is worth noting that in Figures 7 and 8, for edges emanating from conditional nodes, each edge's non-empty label value is simply written next

25   to the edge (rather than being written in the form "label = edge's_label_value"). This alternative notation indicates that, during the course of execution of the SCFG, whether that edge is taken is determined by the edge's label value. In contrast, runningPredecessor and restartPredecessor edges, when a non-empty label value is to be indicated, always have the label value specified by an

30   expression of the form "label = edge's_label_value."

The next step of "Initialization" in Appendix A is the creation of the outermost thread 1001. Figure 6C, line 26. Then thread 1001 is added to the threads of process 1000 (Figure 6C, line 28) and process 1000 is indicated to be the containing process of thread 1001 (Figure 6C, line 30).

5   Next, the first node of the ACCFG, "fork(1)_1002," is obtained. The "1" in "fork(1)_1002" indicates the node's order in the topological sort of the ACCFG, while the "1002" indicates the node's utilization within the processes and/or threads of Figure 8.

The name of the "stateVariable" for thread 1001, as it is to be used in the SCFG, is created. (Figure 6C, line 35) In general, the stateVariable of a thread

10   is a variable used when the SCFG is running that holds the state of the thread when it is suspended.

fork(1)_1002 is added to the "cnodes" of thread 1001. In general, the cnodes of a thread are the nodes of it that are either currently executing or could

15   be executed next.

Thread 1001 is added to the "pthreads" of fork(1)_1002. "pthreads" indicates the threads to which a cnode belongs and are therefore the cnode's "parent threads." As indicated in Figure 6A, lines 9-11, most cnodes belong to exactly one thread. The two exceptions are: the outermost process (which

20   belongs to no threads) and a join node (which belongs to every thread it joins). While not depicted in Appendix A, it is line 22 of Figure 6C which sets the "pthreads" of the outermost process to empty.

Finally, "Initialization" sets the "state" of fork(1)_1002 to "Suspended."

25   ## 2.2 ITERATION 1

Appendix A continues with the heading "1. Main Loop: First Iteration," which indicates that the first iteration of the main loop is begun next. Figure 6D, line 4.

In general, the first iteration is responsible for converting the topologically

30   first node of the input ACCFG into an SCFG node that is properly connected within the SCFG. The first iteration accomplishes this by "resuming" (the

outermost thread is not actually being resumed since it was never previously suspended but was created as suspended by the initialization code) the outermost thread. Whenever a thread is resumed, a restart node is created to test the saved state of the thread being resumed. Since the outermost thread

5   was not actually suspended a trivial restart node (a restart node with only one outgoing edge) is created. Such trivial restart nodes are typically removed when sequential code (such as assembler or the C programming language) is generated from the final SCFG.

In the first iteration, current node "cn" is set to fork(1) (also known as

10   fork(1)_1002). The "1" in the heading indicates the fact of the first MAIN LOOP iteration being performed. All the subsequent headings, that are also indicative of first iteration's execution, are also prefixed with "1." As each major block of code, or each support function, within the first iteration is executed, the "1" is suffixed with additional indicators. For example, since iteration 1 is for a fork

15   node, code blocks B and C are executed. As can be seen, all of the headings within iteration 1 have the "1" suffixed with a "b" (if they represent execution within code block B) or a "c" (if they represent execution within code block C). To illustrate how support functions are indicated in the headings, consider how code block B calls two support functions: "switch to thread th" (Figure 6D, line 33) and

20   "run cnode cn as snode sn" (Figure 6D, line 34). Under the headings of "1.b" when the function "switch to thread th" is called, ".switchTT" is suffixed onto "1.b" and the resulting heading, "1.b.switchTT," is indented to indicate a nesting in the pseudo-code's flow of control. Similarly, ".runCAS" is suffixed onto a heading when "run cnode cn as snode sn" is called. Note that since "switch to thread th"

25   calls "run cnode cn as snode sn" as part of its own execution (Figure 6G, line 27), within the indented heading "1.b.switchTT" is the further indented heading "1.b.switchTT.runCAS."

Under the heading "1. Main Loop: First Iteration," once cn has been assigned the current node of fork(1)_1002, the next to actions are to: i) copy the

30   current node for SCFG, creating current SCFG node 1004 which is assigned to sn (See Figure 8A and Figure 6D, line 6), and ii) obtain the "first" parent thread

1001 to which cn belongs and assign it to th (See Figure 8A and Figure 6D, line 7). For the purposes of the present presentation of pseudo-code simulation, obtaining the "first" parent thread will be defined as obtaining the first parent thread assigned to the cnode of cn. In general, however, any parent thread of
5   the cnode of cn could be obtained and the pseudo-code of Figure 6 would still function correctly.

Since cn is a fork node, code block B is executed next.

The first action of code block B is to call "switch to thread th" with the parameter "th" having been assigned thread_1001. "switch to thread th" operates
10   according to one of the three following scenarios. Before describing these scenarios, it is useful to define the thread "th," that is passed to "switch to thread th," as "thread X," and it is useful to define the process containing thread X as "process Y." Under the first scenario, if thread X is already running, when it is passed as a parameter to "switch to thread th," then "switch to thread th" does
15   nothing. Under the second scenario, if no thread in process Y is running, when thread X is passed as a parameter, then "switch to thread th" creates a restart node for resuming thread X. For the third scenario, if a thread other than X is running in process Y, when thread X is passed as a parameter, then "switch to thread th" first suspends the currently running thread (by calling "suspend any
20   thread running in p") and then, like scenario two, creates a restart node for resuming thread X. In addition to these three basic modes of operation (or scenarios), "switch to thread th" may call itself recursively to insure that higher-level threads, that contain process Y, are also running. Figure 6G, lines 7-10. Note that in the recursive call of "switch to thread th.process.pthreads" the
25   parameter "th.process.pthreads" is specifying only the single parent thread of process Y since, in general, a process can only have one parent thread.

In the case of iteration 1, scenario two is applicable to "switch to thread th." Therefore a restart node 1005 is created for "resuming" thread 1001. Restart node 1005 tests the state of "thread_1001.stateVariable." This is
30   accomplished by the pseudo-code of Figure 6G, line 22. In Figure 8A, "thread_1001.stateVariable" has been abbreviated as "th_1001.stateVar." Like

abbreviations are done throughout Figure 8 for the other restart nodes. In addition to creating the restart node 1005, "switch to thread th" also calls "run cnode p as snode rn" which creates the necessary inbound edges to the restart node in the SCFG. In this particular case, edge 1006 is created. Once "run

5   cnode p as snode rn" has been run, Figure 8A is no longer applicable since edge 1065 is removed by "run cnode p as snode rn." The continued execution of "switch to thread th" is therefore understood by reference to Figure 8B.

Once the execution of "switch to thread th" has completed, code block B next calls "run cnode cn as snode sn." Figure 6D, line 34. This invocation of

10   "run cnode cn as snode sn" creates the correct inbound edges to node 1004 from the rest of the SCFG. In this particular case, the invocation creates the edge 1066 from the restart node with a label of "1." Although the value of thread_1001.stateVariable will not be set when control reaches this restart node when the SCFG is running, edge 1066 will always be taken since it is the only

15   outgoing edge from node 1005. In general, conditional nodes with only one outgoing edge are optimized away.

Since fork(1)_1002 has been "run" (i.e., all the inbound edges to its SCFG equivalent, node 1004, have been created), code block B concludes by removing fork(1)_1002 from thread 1001. Therefore, the continued execution of iteration 1

20   should be viewed in conjunction with Figure 8C.

Code block C of iteration 1 is executed next. As discussed above, code block C is only executed for fork nodes. In general, code block C does the following. It creates a process for the fork node. Figure 6E, line 3. It creates a thread for each outgoing edge of the fork node (Figure 6E, line 13) and each

25   such thread is put in the fork's process (Figure 6E, line 14). The destination node of each of the fork's outgoing edges is added to the appropriate newly created thread. Figure 6E, lines 18-19. The process for the fork starts out runnable (Figure 6E, line 4), but all of its threads are suspended (Figure 6E, line 5). The next iteration of the main loop, which makes a destination node of one

30   of the fork's outgoing edges the current node, will resume the appropriate thread of the fork's process.

For the specific example of Figure 7, under the heading of "1.c" of Appendix A, the following occurs. A process 1008 is created for fork(1). For successor conditional_A(2)_1016 a new thread 1009 is created. For successor conditional_B(4)_1017 a new thread 1010 is created. Process 1008, thread 1009 and thread 1010 are all depicted in Figure 8C.

5

2.3 ITERATION 2

For the specific example of Figure 7, under the heading of "2." of Appendix A, the actions of the MAIN LOOP can be summarized as follows. The second node of the topological ordering, conditional_A(2)_1016 is "run" (meaning that, after it is copied to the SCFG, its SCFG node has the correct inbound edges connected to it). This entails resuming the thread containing conditional_A(2)_1016, thread 1009, which therefore entails the creation of another trivial restart node (node 1012). The restart node 1012 is trivial (i.e., has only one outbound edge) because thread 1009 was never previously suspended, but this lack of distinction between previously and non-previously suspended threads simplifies the pseudo-code of Figure 6.

The step-by-step operation of the second iteration is as follows.

As with iteration 1, the first three actions are to: i) make conditional_A(2)_1016 the current node "cn"; ii) make a current SCFG node 1011 that is a copy of conditional_A(2)_1016; and iii) obtain the parent thread 1009 containing current node conditional_A(2)_1016.

As with fork(1)_1002, the next step is to execute code block B since a conditional node is a "normal" node.

The first step of code block B is to execute "switch to thread thread_1009."

Since thread_1009 is not the outermost thread, the first action of "switch to thread" is to recursively call itself, with outermost thread_1001 as the input parameter. This recursive call to "switch to thread" (under the heading "2.b.switchTT.switchTT" of Appendix A) verifies that thread_1001 is already running and returns.

Next, "switch to thread thread_1009" performs scenario 2 by creating a restart node 1012 which tests the state of "thread_1009.stateVariable." "switch to thread thread_1009" also creates the correct inbound edge to restart node 1012 by calling "run cnode process_1008 as snode rn_1012" (as can be

5 seen under the heading "2.b.switchTT.runCAS" of Appendix A). Once this invocation of "run cnode process_1008 as snode rn_1012" has completed, it is appropriate to consider the further execution of "switch to thread thread_1009" in conjunction with Figure 8D.

The main remaining actions of "switch to thread thread_1009" are as

10 follows. Restart node 1012 is established as the restartPredecessor of conditional_A(2)_1016 and thread_1009 is set to be the "Running" thread within process_1008.

The second function call of code block B, "run cnode conditional_A(2)_1016 as snode sn_1011." This "running" of sn_1011 (the node

15 in the SCFG that represents conditional_A(2)_1016 of the ACCFG) causes the correct inbound edge to sn_1011 (edge 1018) to be created. Note that edge 1018 has the label "2." While this label would seem to indicate that the state variable for thread_1009 (i.e., thread_1009.stateVariable) must be initialized to the value "2" when the SCFG is executed to enable edge 1018 to be taken, since

20 1018 is the only edge it will always be taken regardless of the value of state variable.

Finally, code block B removes conditional_A(2)_1016 from thread_1009 since it has been converted into the SCFG.

At this point, it is appropriate to consider the continued execution of

25 Appendix A in conjunction with Figure 8E.

Since current node conditional_A(2)_1016 is a normal cnode, rather than a fork cnode of the first iteration, code block D is executed next. Code block D iterates over each successor to conditional_A(2)_1016 in the ACCFG and adds it to thread_1009. It also makes sn_1011 the "runningPredecessor" of each

30 successor to conditional_A(2)_1016 so that these successors will have the

correct inbound edges generated in later iterations (specifically in iteration 3 for successor emit_B(3)_1021 and iteration 8 for join(8)_1019).

2.4 ITERATION 3

Iteration 3, in which emit_B(3)_1021 is the current node, is executed next. Of the total of 8 iterations of the MAIN LOOP performed in processing the example of Figure 7, iteration 3 is one of the simplest. This is because emit_B(3)_1021 is in a thread that is already running, so no context switching is necessary. The major actions of iteration 3 are to: i) make an SCFG copy of emit_B(3)_1021, this being node sn_1023; ii) attach an inbound edge 1024 to sn_1023 (based on conditional_A_1011 being the single runningPredecessor to emit_B(3)_1021); and iii) make the single successor to emit_B(3)_1021, conditional_C(6)_1025, have sn_1023 as its runningPredecessor.

Iteration 3 is similar to iteration 2 in that both are handling a current node "cn" of the normal type. Therefore, for both iterations, code blocks B and D are executed. The execution of "switch to thread thread_1009", in code block B, is much simpler for iteration 3 because thread_1009 is already running. The execution of code block D is similar in both iterations 2 and 3, except that in iteration 3 there is only one successor of "cn".

2.5 ITERATION 4

Iteration 4, in which "cn" is conditional_B(4)_1017, is one of the most complex of the 8 iterations since a context switch, which saves the state of thread_1009, must be performed. Thread_1009 is suspended by adding state-saving nodes 1028 and 1031. Thread_1010 is "resumed" by adding a trivial restart node 1034, which has a single edge going to sn_1027 (the SCFG copy of conditional_B(4)_1017).

Iteration 4 is similar to iterations 2 and 3 in that code blocks B and D are executed. The execution of code block B differs in iteration 4 since scenario 3 is performed. This means that within the execution of "switch to thread

thread_1010," an execution of "suspend any running thread in process_1008" is performed (see heading "4.b.switchTT.suspendART" of Appendix A).

The execution of "suspend any running thread in process_1008" should be considered in conjunction with Figures 8F through 8H. This function 5 suspends thread_1009 which has two runnable cnodes: conditional_C(6)_1025 and join(8)_1019. To distinguish between whether thread_1009 is being suspended with conditional_C(6)_1025 as the next to be executed (actually the SCFG copy of conditional_C(6)_1025 as the next to be executed) or with join(8)_1019 as the next to be executed, the state-saving nodes of, respectively, 10 1031 and 1028 are added.

State-saving node 1028 is given an assignment statement that stores an "8" in the stateVariable for thread_1009, while state-saving node 1031 is given an assignment statement that stores a "6." State-saving node 1028 is given in-bound edge 1029 since join(8)_1019 is a runningPredecessor of 15 conditional_A_1011. State-saving node 1031 is given in-bound edge 1032 since conditional_C(6)_1025 is a runningPredecessor of conditional_A_1011.

When "switch to thread thread_1010" resumes, after "suspend any running thread in process_1008" completes, restart node 1034 is created (see Figure 8H). Then "switch to thread thread_1010" executes "run cnode 20 process_1008 as snode rn_1034" which adds the correct inbound edges to restart node 1034 from the two state-saving nodes (see Figure 8H).

Once "switch to thread thread_1010" is completed, code block B executes "run cnode conditional_B(4)_1017 as snode sn_1027" which creates the single in-bound edge 1038 to sn_1027 (also known as conditional_B_1027). See 25 Figure 8I. While edge 1038 is given the label "4," meaning that thread_1010.stateVariable would appear to require initialization to the value 4 when the SCFG is to be executed, since 1038 is the only edge, it is always taken regardless of the value of the state variable.

Once code block B is completed, code block D is executed. See heading 30 "4.d" of Appendix A and Figure 8J. Code block D places the successors of conditional_B(4)_1017 in thread_1010. The result of this, as can be seen from

Figure 8J, is to put join(8) in two threads: thread_1009 and thread_1010. Therefore, the "pthreads" property of join(8) will have both of these threads. Notice that while it is the same data object, join(8) is shown as join(8)_1019 in thread_1009 and as join(8)_1039 in thread_1010 in order to emphasize its

5   location within two different threads.

2.6 ITERATION 5

Iteration 5, in which the current node is emit_C(5), is a simple iteration like iteration 3. Like iteration 3, this simplicity is due to the fact that no context

10   switching is necessary: thread 1010 is already running and emit_C(5) is simply added to that thread.

2.7 ITERATION 6

Iteration 6 is like iteration 4 in that it also has a context switch. This

15   switching between threads (from thread 1010 as running to thread 1009) means that scenario 3 of "switch to thread" is performed which, in turn, means that "suspend any running thread" is executed. The operation of "suspend any running thread" is simpler in iteration 6, than in iteration 4, since state-saving nodes are not needed (because thread 1010 will always resume at join(8)).

20   "suspend any running thread in process_1008" makes conditional_B_1027 and emit_C_1043 runningPredecessors of process_1008. Therefore, when "switch to thread thread_1009" is subsequently resumed, the restart node it creates (node 1047) is given inbound edges from conditional_B_1027 and emit_C_1043 (See Figure 8L). As part of code block B, iteration 6 also creates the edge 1052

25   from restart node 1047 to sn_1045. Edge 1052 has been appropriately labeled with "6." Restart node 1047 is the first, and only, non-trivial restart node resulting from the example of Figure 7. Iteration 6 ends with code block D which adds the successors to conditional_C(6)_1025 to thread 1009 (See Figure 8N).

30   ## 2.8 ITERATION 7

Iteration 7, in which the current node is emit_D(7), is a simple iteration like iterations 3 and 5. Like iterations 3 and 5, this simplicity is due to the fact that no context switching is necessary: thread 1009 is already running and emit_D(7) is simply added to that thread.

5

2.9 ITERATION 8

Iteration 8 is the final iteration of the MAIN LOOP in which join(8) is converted into the SCFG. The fact that join(8) resides in two threads does not result in ambiguity: either thread can be selected by the pseudo-code at Figure 6D, line 7. Unlike any other cnode type, a join node causes the MAIN LOOP to execute code block A. As can be seen, code block A simply uses the thread selected to identify its containing process. For either of threads 1009 or 1010, the containing process is 1008. As discussed above, process 1008 was created to represent join(8) and it is process 1008 that is "run" by code block A. Process 1008 is "run" in its parent thread 1001. Code block A executes a call to "switch to thread thread_1001" to confirm that outermost thread 1001 is running. The next call by code block A, to "suspend any running thread in process_1008," suspends thread 1009. "suspend any running thread in process_1008" converts the runningPredecessors and restartPredecessor of join(8)_1019 into runningPredecessors of process 1008. See Figure 8P where edges 1051, 1052 and 1057 become, respectively, edges 1059, 1060 and 1061. Once this is done, code block A is then able to "run" process 1008 as snode 1058, which creates all the necessary remaining edges for the SCFG. The SCFG edges created are, namely, 1062 (for 1059), 1063 (for 1060) and 1064 (for 1061).

Since join(8) has no successors, code block D does nothing in iteration 8.

It should be noted that in Figures 8J and 8K, while only fork(8) 1039 is shown as having edges 1040 and 1067, this is for illustrative purposes and, in fact, fork(8) 1019 has the same edges. Similarly, it should be noted that in Figures 8M through 8P, that while only fork(8) 1019 is shown as having edges 1051, 1052 and 1057, this is for illustrative purposes and, in fact, fork(8) 1039 has the same edges.

Since all 8 iterations of the MAIN LOOP have completed, "synthesize a scfg" returns with the SCFG shown on the right side of Figure 8Q as its final product.

3. HARDWARE ENVIRONMENT

Typically, the conversion architecture of the present invention is executed within the computing environment (or data processing system) such as that of Figure 9. Figure 9 depicts a workstation computer 900 comprising a Central Processing Unit (CPU) 901 (or other appropriate processor or processors) and a memory 902. Memory 902 has a portion of its memory in which is stored the software tools and data of the present invention. While memory 903 is depicted as a single region, those of ordinary skill in the art will appreciate that, in fact, such software may be distributed over several memory regions or several computers. Furthermore, depending upon the computer's memory organization (such as virtual memory), memory 902 may comprise several types of memory (including cache, random access memory, hard disk and networked file server). Computer 900 is typically equipped with a display monitor 905, a mouse pointing device 904 and a keyboard 906 to provide interactivity between the software of the present invention and the chip designer. Computer 900 also includes a way of reading computer readable instructions from a computer readable medium 907, via a medium reader 908, into the memory 902. Computer 900 also includes a way of reading computer readable instructions via the Internet (or other network) through network interface 909. The software tools and data of the present invention may be stored as computer readable instructions on a computer readable medium, such as 907. The software tools and data of the present invention may also be transported into a computer system over a network and through a network interface, such as 909. Such network transmission may involve the use of a carrier wave.

As a target architecture for the present invention, upon which the sequential code produced from the SCFG would be executed, the system of Figure 9 stores the target sequential code in region 903 of memory 902. The

target architecture of the present invention can be substantially simpler than the architecture shown in Figure 9. Specifically, one or more of the user interface components, such as 904, 905 and 906, are often not necessary. The target architecture of the present invention would often be an embedded system which

5　could be used in a variety of applications such as: a wristwatch, a cellular telephone or the fuel injection of an automobile. Rather than the user interface components, such embedded systems would often include a variety of different sensor and/or actuator peripheral devices for interfacing the computing system with its operating environment. Element 910 of Figure 9 represents, generally,

10　any one of a variety of sensory input devices which might be used. Element 911 of Figure 9 represents, generally, any one of a variety of actuator output devices which might be used.

　　　　While the invention has been described in conjunction with specific

15　embodiments, it is evident that many alternatives, modifications and variations will be apparent to those skilled in the art in light of the foregoing description. Accordingly, it is intended to embrace all such alternatives, modifications and variations as fall within the spirit and scope of the appended claims and equivalents.